HDL RISC-V-A Project Report

Nour Hadj Fredj, Mariia Kashirina, Roman Mishchuk, Mohamed Amine Jradi
Prof. Dr. Carsten Trinitis*
**Chair of Computer, Information and Technology,*
Department of Informatics
Technical University of Munich, Heilbronn, Germany
{*nour.hadj-fredj, mariia.kashirina, roman.mishchuk, amine.jradi, carsten-trinitis*}*@tum.de*

1. Introduction

In our project, we investigate how a 16-bit RISK-V architecture works. We explore how simple arithmetic operations like multiplication and division are performed, their synthesis in combinatorial logic, and how other parts of the processor interact with each other. We have done work to adjust the default structure of the RISK-V processor so that it matches the given specifications. This report will give the reader a better understanding and an overview of RISK-V architecture. The source code to the Verilog implementation of it and its test benches can be found online[1].

1.1. Notation and terminology

- R_i – the i-th register of the processor;
- `MonoFont` – the indication of a code snippet, parameter name of a keyword;
- `l` and `lv` – parameters used throughout our code to specify the bitness of the parameters. By default, `l` = 16 (given that we are working with a 16-bit processor), and `lv` = 15 (the highest bit index of a 16-bit number). But we can decrease them for the testing purposes.

2. Specifications

We implemented a RISC-V module that consists of Instruction Memory, Control Unit, Register File (16-bit register length, 8 registers in total), and ALU. The usual Data Memory is removed, and results go back to the Register File, but the order and concrete Instructions themselves are hard-coded in the Instruction Memory module.

An Instruction is 16-bits long and can be understood in three different ways, depending on the operation we want to perform. Following Little Endian, bits [15:13] always stand for the opcode, and the next [12:10] bits represent the address of Register where the result of the operation will be stored. Then, for RRR-operation, bits [9:7] and [2:0] represent addresses of registers to fetch the data from. For RRI-operation, bits [9:7] are set for the register address, and the last [7:0] bits represent the immediate signed value. For LUI operation, we only need the resulting register and the immediate value to put to this register, so all [9:0] bits represent this value.

Program Counter is implemented and connected to the Instruction Memory where, depending on its value, it chooses a pre-coded instruction. The processor is able to perform multiplication (16-bit, RRR and RRI), division (16-bit, RRR and RRI), and load values to Registers. Everything is happening in a single cycle: the instruction is fetched from the Instruction Memory, the Control Unit receives an opcode and sends corresponding signals, ALU executes the instruction, and results come back to the Register File.

Processor implementation is also extended by creating a Flag Register, which is not updated every Clock Cycle (Control Unit sends a special Update-Flags signal if something needs to be written to Flag Register).

TABLE 1: Flags Register reference

index	Description
0	Multiplication overflow
1	Division has remainder
2	Division by zero
3	Division overflow $(-2^{(16-1)} \cdot -1)$

3. Instruction Memory

The instruction Memory module stores the read-only instructions to be executed on the processor. The following subsections describe this module's design, implementation, and testing.

3.1. Instruction Memory design

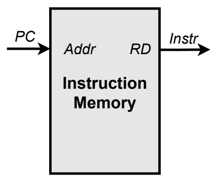

Figure 1: Instruction Memory design

- `Addr` – the address of the instruction to be read;

1. https://github.com/superstudy160/RISC-V-i16-A-Project.

- `Instruction` – the instruction read from the memory at the specified address.

3.2. Instruction Memory implementation

Since this memory is read-only, we hard-coded its contents in the module definition.

Data addressing is done byte-wise. Thus, given that instructions are 2 bytes long, we omit the first bit of the address when selecting the output (and so outputting the same instruction for both even and odd addresses).

We implemented such a behavior using the `case` statement:

Listing 1: Hard-coded Instruction example

```
always @(Address) begin
case (Address[lv:1]) // Here we omit the first bit
  // This is the contents of the instruction memory
  00: Instruction = { 3'b011, 3'b000, 10'sd2 };
  ...
```

If the address goes out of the bounds of the memory, we return the default command, which does the multiplication of the R_0 by 1:

Listing 2: Default Instruction

```
  ...
  default: Instruction = { 3'b001, 3'b000, 3'b000, 7'sd1 };
endcase
...
```

4. Register File

The Register File serves as a source for operands and a destination for storing results during arithmetic or logical operations. In instruction execution, the processor reads operands from the register file, performs operations, and writes results back to it.

4.1. Overview and Code implementation

We see that the Register File has as an input a clock signal, 3 addresses, InDataA, UpdateFlags and InNewFlags. For outputs we have We have OutDataB and C, OutFlags, and we also added a DebugData wire that we used to test our module and show the content of the registers. We have declared 3 Addresses for our input because our simplified RISC-V processor has 3 formats for instruction where RI type calls only one register, RRI type calls 2 registers and RRR types calls 3.

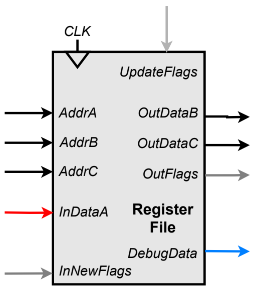

Figure 2: Register file diagram

After the input and output declaration, we specified that R_7 – flags register using the following parameter:

Listing 3: Flags Register address

```
parameter FlagsAddress = 7;
```

Then we declared an array of registers named Data. This array consists of eight 16-bit registers which will be initialized to zero.

This part is concerned with the read functionality of the Register File where we assign the data stored at the address specified by AddrB to OutDataB and we do the same for AddrC and OutDataC. We also assign the data stored in FlagsAddress to OutFlags. This allows us to read the content of the registers.

Listing 4: Read functionality

```
assign OutDataB = Data[AddrB];
assign OutDataC = Data[AddrC];
assign OutFlags = Data[FlagsAddress[2:0]];
```

On the other hand, this section is responsible for the write functionality where on the positive edge of the clock signal, the block triggers and writes the input data InDataA into the register specified by AddrA. Also, if UpdateFlags is asserted, it updates the flags data stored at the address specified by FlagsAddress with the new flags data InNewFlags.

Listing 5: Write functionality

```
always @ (posedge Clk) begin
   Data[AddrA] = InDataA;

   if (UpdateFlags)
   Data[FlagsAddress[2:0]] = InNewFlags;
end
```

This final part has been added mainly for debugging and testing our Register File.

Overall, our module implements a simplified register file with read and write functionality, flags update capability, and debug output for monitoring register content while operating synchronously with the clock signal.

5. Control Unit

The control unit interprets instructions (opcodes) from the Instruction Memory and decodes them to determine what operations are to be performed. In the following subsections, we will discuss the Control Unit's behavior and how it is implemented.

5.1. Behavior of Control Unit

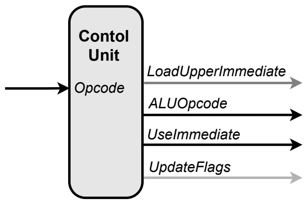

Figure 3: Interface of the Control Unit

When receiving an `Opcode`, the Control Unit outputs four different signals:

- `LoadUpperImmediate`, which tells whether we need to store data from ALU or instruction (to execute LUI instruction);
- `ALUOpcode`, which sends ALU 0 or 1 depending on the type of operation we are performing (multiplication, division);
- `UseImmediate`, which tells whether the C operand is an immediate value or the value from the register file;
- `UpdateFlags`, which indicates whether the Register File should update flags after the operation or not.

An overview of the input and output signals of the Control Unit is shown in Figure 3.

5.2. Implementation of Control Unit

We have five different opcodes to be handled; they are 3 bits long and stored in [15:13] bits of the instruction. In the following, we will discuss the behavior of the Control Unit for each of these opcodes.

Load Upper Immediate (LUI, 011)

The dummy signal is sent to all inputs of ALU because there is no computation to do.

- `ALUOpcode` = 0 (dummy signal);
- `UseImmediate` = 0 (dummy signal);
- `LoadUpperImmediate` = 1 – in this case, we ignore ALU output and take the immediate value from the instruction;
- `UpdateFlags` = 0 – there is no flags to update.

In all the following cases, the upper immediate value is ignored; thus, `LoadUpperImmediate` is set to 0.

Multiplication (MUL, 111)

- `ALUOpcode` = 1 (multiplication);
- `UseImmediate` = 0 (RRR operation);
- `LoadUpperImmediate` = 0;
- `UpdateFlags` = 1 – to renew the flags' state.

Multiplication Immediate (MULi, 001)

- `ALUOpcode` = 1 (multiplication);
- `UseImmediate` = 1 (RRI operation);
- `LoadUpperImmediate` = 0;
- `UpdateFlags` = 1 – to renew the flags' state.

Division (DIV, 000)

- `ALUOpcode` = 0 (division);
- `UseImmediate` = 0 (RRR operation);
- `LoadUpperImmediate` = 0;
- `UpdateFlags` = 1 – to renew the flags' state.

Division Immediate (DIVi, 010)

- `ALUOpcode` = 0 (division);
- `UseImmediate` = 1 (RRI operation);
- `LoadUpperImmediate` = 0;
- `UpdateFlags` = 1 – to renew the flags' state.

In the code, this logic is implemented using the case keyword:

Listing 6: Opcode switch-statement example

```
always @(Opcode) begin
case (Opcode)
   // MUL
   3'b111: begin
      ALUOpcode = 1;
      UseImmediate = 0;
      LoadUpperImmediate = 0;
      UpdateFlags = 1;
   end
...
```

6. ALU

In the section we will focus on the ALU in the RISC-V architecture by explaining in detail how it works. The ALU is the arithmetic logic unit and in the context of this project, it will only do multiplication and division operations.

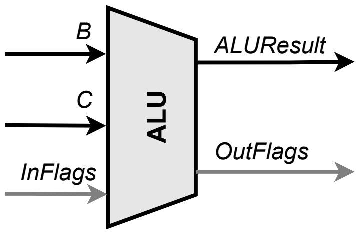

Figure 4: ALU overview

6.1. Overview

For an operation to be executed, the ALU receives a bit value from the control unit to indicate which operation to execute. As for the inputs, the ALU receives from the register file with the flags. For both operations, the ALU will first change the inputs to their absolute values.

Listing 7: Assigning Absolute value operation

```
wire [lv:0] AbsA;
AbsoluteValue #(l) abs_a(
   .A(A), .R(AbsA)
);
wire [lv:0] AbsB;
AbsoluteValue #(l) abs_b(
   .A(B), .R(AbsB)
);
```

According to the operation intended, the ALU will import the flags necessary. In case of multiplication, the ALU will update the value of multiplication overflow flag.

Listing 8: Assigning multiplication operation

```
1: begin
  // Multiplication
  UnsignedR = MultiplicationR;
  FlagsOut[MultiplicationOverflowIdx[lv:0]] =
  MultiplicationOverflow | SignOverflow;
  FlagsOut[DivisionHasRemainderIdx[lv:0]] =   FlagsIn[
        DivisionHasRemainderIdx[lv:0]];
  FlagsOut[DivisionByZeroIdx[lv:0]] = FlagsIn[
        DivisionByZeroIdx[lv:0]];
  FlagsOut[DivisionOverflowIdx[lv:0]] = FlagsIn[
        DivisionOverflowIdx[lv:0]];
end
```

In our case when the operation wire has 1 then that indicates that a multiplication operation should be executed. In the case of a division, there are two flags to be updated: divisionHasRemainder, which will indicate a remainder, and divisionDivByZero, which will indicate the edge case of division by zero.

Listing 9: Assigning division operation

```
0: begin
// Division
UnsignedR = DivisionR;
FlagsOut[DivisionHasRemaindeFlagOut[lv:0]] =
      DivisionHasRemainder;
FlagsOut[DivisionByZeroFlagOut[lv:0]] = DivisionDivByZero;
FlagsOut[DivisionOverflowFlagOut[lv:0]] = SignOverflow;
FlagsOut[MultiplicationOverflowFlagOut[lv:0]] = FlagsIn[
      MultiplicationOverflowFlagOut[lv:0]];
end
```

In case of no flags are raised the noFlagsIdx. After doing the operation between two unsigned integers. The ALU operates to give a sign to the result.

Listing 10: Sign operation

```
reg [lv:0] UnsignedR;
wire SignOverflow;
GiveSign #(l) give_sign(
   .Sign(A[lv] ^ B[lv]),
   .A(UnsignedR),
   .R(R),
   .Overflow(SignOverflow)
);
```

The overall structure can be seen in Figure 5:

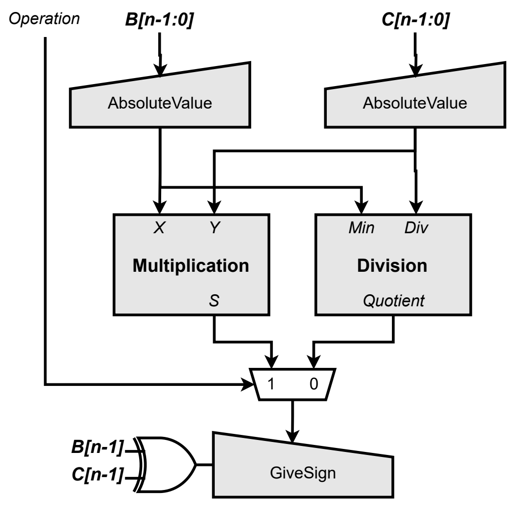

Figure 5: ALU structure

6.2. Addition

Listing 11: 1 bit adder

```
module Adder (
   input X,
   input Y,
   input Cin,
   output S,
   output Cout
);

assign S = X ^ Y ^ Cin;
assign Cout = (X & Y) | ((X ^ Y) & Cin);

endmodule //Adder
```

We start by implementing one bit adder. Then we use it to implement a full adder for 16 bits.

Listing 12: Full adder

```
module FullAdder #(parameter l = 16) (
   input [lv:0] X,
   input [lv:0] Y,
   input Cin,
   output [lv:0] S,
   output [lv:0] Cout
);

parameter lv = l-1;

wire [lv:0] Sum;
wire [l:0] Cout_temp;
assign Cout_temp[0] = Cin;

// Remaining bits
generate genvar i;
   for (i = 0; i ⩽ lv; i = i + 1) begin
      Adder adder(X[i], Y[i], Cout_temp[i], Sum[i], Cout_temp[
            i+1]);
   end
endgenerate

assign S = Sum;
assign Cout[lv:0] = Cout_temp[l:1];

endmodule //FullAdder
```

We reimplement this full adder to make a full adder with overflow and carry flag. Overflow in case the addition of two positive number became negative because the range of bits representing the integers was exceeded. Carry in case the range of bits to represent the number was exceeded.

Listing 13: Full adder with flags

```
module FullAdderFlags #(parameter l = 16) (
   input [lv:0] X,
   input [lv:0] Y,
   output [lv:0] S,
   output Overflow,
   output Carry
);

parameter lv = l-1;

wire [lv:0] Cout;

FullAdder #(l) full_adder (
   .X(X), .Y(Y), .Cin(1'b0),
   .S(S), .Cout(Cout)
);

assign Overflow = Cout[lv-1] ^ Cout[lv];
assign Carry = Cout[lv];

endmodule // FullAdderFlags
```

6.3. Multiplication

The multiplication is a composition of the addition operation and shift. That was the approach used, make the multiplication operation through the addition block we created. In order for the multiplication to work we just copy the multiplier whenever we find a 1, then shift or skip and shift when we find a zero. We only add carries in the next operation for optimization purposes. So we do an addition operation then we add the result to the next addition operation and we add the carry too. That is the general concept. However, to optimize this operation, each time we shift we add the bits on the right to the final result and we keep going until the operation is finished. The result will be represented in 32 bit but we only compute the first 16 bits the second half is discarded and if it has some value in it then it is an overflow. These operations will be further explained one by one in the next sub-subsections. Of course there are certain edge cases like carry and overflow which have specific flags to be raised when they happen.

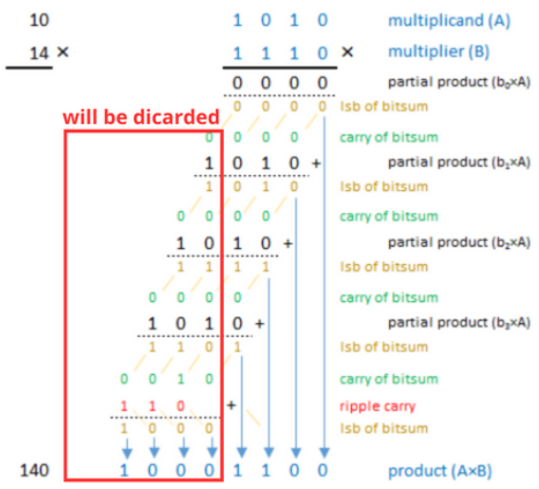

Figure 6: Multiplication

6.3.1. Contolled adder circuit. The controlled adder is the same as the 1 bit adder. As a matter of fact it uses the 1 bit adder we created only with one twist. we add another input to this controlled adder which is selection. This input will specify if we preceed with the addition or we ignore it and shift. In our case if the inpur is 1, we will preceed with the addition and if it is 0 we skip.
The figure below illustrate the diagram of controlled adder, as x, y, cin and selection are the inputs and cout and S are the outputs.

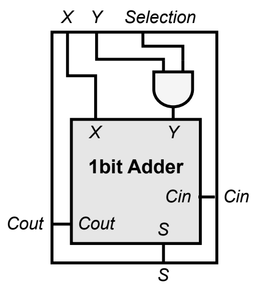

Figure 7: Controlled adder diagram

6.3.2. Contolled full adder circuit. The controlled full adder circuit is the collection of the controller adder, which will allow us to do the same operation for 16 bits. So we just use the controlled adder multiple time. As for the carry we do not care about it as it will be passed to the next full adder. In other words cout for each controlled adder will not be connected to the next one rather to the one below it as shown in 3.3.3. This way we are multiplying one bit from the multiplier with all bits of the multiplicand.

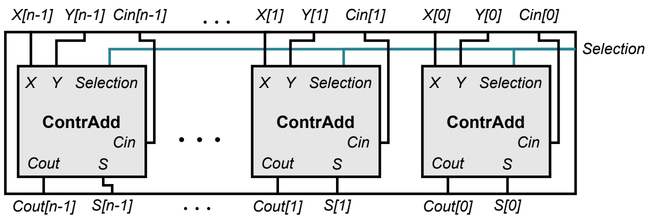

Figure 8: Controlled full adder diagram

6.3.3. Multiplication circuit. Now we need to do the same operation of the controlled full adder 16 times. This way we are multiplying each bit of the multiplier with all bits of the multiplicand. The result will be 32 bits. That is why we only compute the first 16 bits as we consider any bit out of that range overflow.
As shown in the simplified diagram below, if we are dealing with a 3 bit operation. The result will be 6 bits but we only compute the first 3 and we take the rest of the bits and the carry, we compare it with 0 to decide if there is a carry or an overflow.

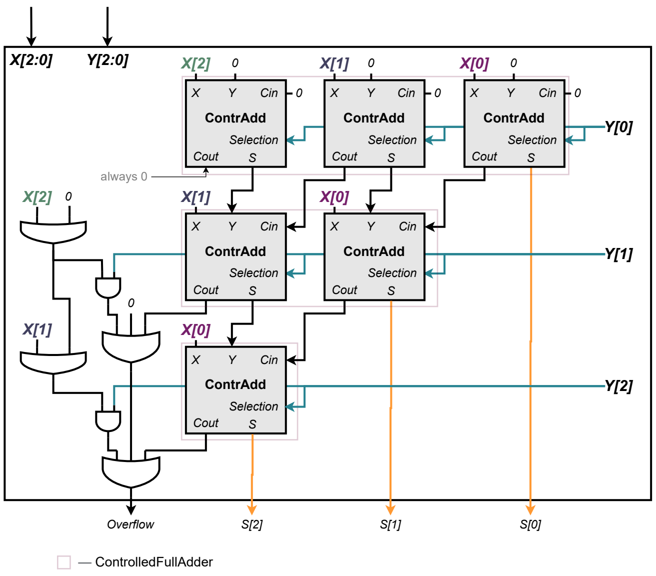

Figure 9: Multiplication diagram

6.4. Division

The division works similarly to the multiplication but with some changes. The division is a composition of the subtraction and shift operation. That was the approach used, make the division operation through the subtraction block we created. In order for the division to work we just copy the divider whenever we find a 0 in the borrow, then shift or skip and shift when we find a 1. So we do a subtraction operation then we shift and add the result to the next subtraction operation. That is the general concept. We created flags for division by zero and remainder. So for each case, a flag will be raised. The division will be discussed in more detail in the next sub-subsections.

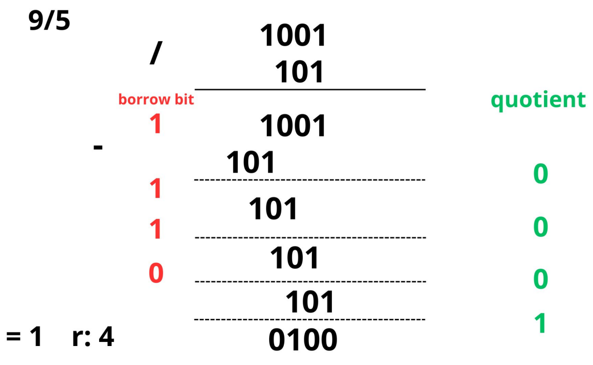

Figure 10: Division

6.4.1. Subtractor. This block is to subtract one bit from another. To implement the subtraction we use the following formulas:

$$D = X \oplus Y \oplus \mathrm{Bin}$$
$$Bout = (\neg X \wedge Y) \vee ((\neg(X \oplus Y)) \wedge \mathrm{Bin})$$

6.4.2. Controlled subtractor. The controlled subtractor, just like the controlled addition, is the same as a subtractor but with a small addition, which is the Selection input as shown in the diagram. Unlike the controlled addition, when the selection input is 0, the difference will be output as the difference. However when the selection is 1, the minuend will be out as the difference.

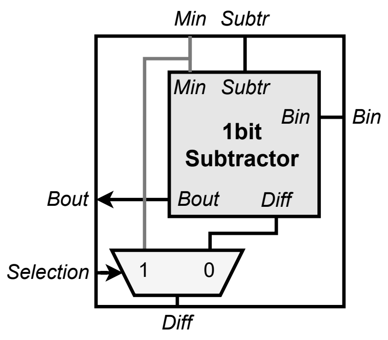

Figure 11: Controlled subtractor diagram

6.4.3. Controlled full subtractor. The controlled full substractor in the collection of controlled subtractors. They are linked through the borrow out and borrow in as each borrow out becomes the borrow in of the next block. As shown in the figure, the output of the borrow out (Bout) will be forwarded to selection to determine if the output of the difference will be the difference or the minuend as explained in the previous sub-subsection. The Bout of the most significant bit is negated to form the quotient. If there is no borrow then the subtraction is possible and then the quotient bit in that block is 1 if not the the bit is 0.

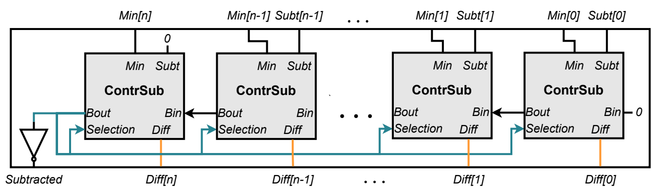

Figure 12: Controlled subtractor diagram

6.4.4. Division. Finally, to form the division operation, the combination of multiple layers of controlled full subtractor will result in the division operation. In the Figure 13 below, we can see a simplified division operation for two 3-bit integers. It is clear how the shift happens from layer to layer and within each layer, a bit from the minuend is included as input. the last block in the first layer is not an input block but rather a block so the most significant bit finds from where to borrow. The difference is forwarded from layer to layer bit by bit until it reached the end of the division where it will be forwarded to hasRemainder. If it is not 0 then the flag will be raised. Also if the divider is 0 it will also raise DivByzero. That is why we have NOR to check if we have a 000 or not.

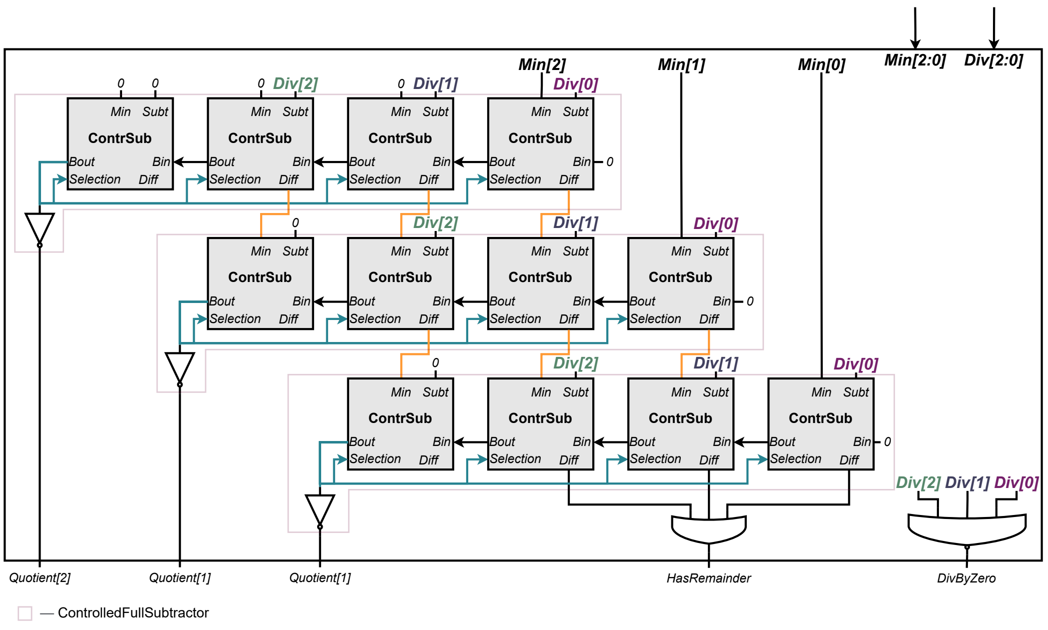

Figure 13: Division diagram

7. Debugging and testing

In this section, we will go through the test benches of each operation.

7.1. Instruction Memory test bench

In the test bench, we iterated through all the memory addresses (including the out-of-bounds ones) and checked if the output was as expected:

```
initial begin
    $monitor(”address=%d, instruction=%b\n”, Address,
        Instruction);
    for (Address = 0; Address < 40; Address = Address + 1)
        begin
      #10;
    end
end
```

During the testing, we did not find any issues.

7.2. Register File Test Bench

This final part has been added mainly for debugging and testing our Register File.

Listing 14: Testing and Debugging

```
    generate
      genvar i;
      for (i = 0; i < r; i = i + 1) begin
      assign DebugData[i * l +: l] = Data[i[2:0]];
      end
    endgenerate
```

7.3. Control Unit Test Bench

In our implementation of the test bench, we went through all the possible opcodes and checked if the module output was correct.

Listing 15: Test Bench of Control Unit

```
initial begin
    $monitor(”LoadUpperImmediate=%d\nALUOpcode=%d\nUseImmediate
        =%d\nUpdateFlags=%d\n\n”, LoadUpperImmediate,
        ALUOpcode, UseImmediate, UpdateFlags);

    $display(”MUL:”);
    Opcode = 3’b111;
    #10;
    ...
```

During the testing, we didn’t find any issues.

7.4. Addition Test Bench

In implementing the test bench, we went through all the possible edge cases, which are overflow, carry or overflow and carry at the same time.

Listing 16: Addition Test Bench

```
initial begin
    $monitor(”  a=%b\n  b=%b\nsum=%b, o=%b, c=%b\n\n”, a, b,
        sum, overflow, carry);
  ...

      x = 16’b1000000000110000;
      y = 16’b1000000011100000;
        #10;
  ...
      x = 16’b0111111111111111;
      y = 16’b0000000000000001;
      #10;
      x = 16’b1111111111111111;
      y = 16’b0000000000000001;
        #10;
    end
    ...
```

7.5. Multiplication Test Bench

In implementing the test bench, we went through all the possible edge cases, which is overflow. To make the overflow easier to detect we used in our test bench 3-bit integers.

Listing 17: Multiplication Test Bench

```
initial begin
...
    $monitor(”X=%d,\nY=%d\nR1=%d, O=%b\n”, X, Y, R1,
         Overflow);
 ...
    X = 16’d3;
    Y = 16’d4;
      #10;
  ...
```

7.6. Division Test Bench

In implementing the test bench, we went through all the possible edge cases, which is divByzero and remainder.

Listing 18: Division Test Bench

```
  initial begin
    $monitor(”min=%d\ndiv=%d\nq=%d, r=%b, z=%b\n\n”, min,
         div, quotient, HasRemainder, DivByZero);
    min = 16’d0;
    div = 16’d0;
    #10;
 ...
    min = 16’d7;
    div = 16’d0;
    #10;
    min = 16’d5;
    div = 16’d2;
      #10;
  ...
```

7.7. ALU Test Bench

In implementing the test bench, we went through all the possible edge cases of multiplication and division to make sure that all modules are connected and work correctly.

8. RISC-V module

We have explored how each module of the processor is implemented, and in this part of the report, we will show how they interact with each other. The design of the module is shown in Figure 14.

8.1. RISC-V execution flow

Every new clock cycle, the Program Counter (PC) is increased by 2, which leads to the new Instruction being fetched from the Instruction memory.

Next, the bits [15:13] of the Instruction are loaded into the Control Unit as the Opcode. If the Opcode corresponds to LUI operation, then `LoadUpperImmediate` output would be set to 1 and `UpdateFlags` to 0. This would lead to the MUX controlling the Register File input, sending the data from ImmediateExtend, and the Register File not accepting the NewFlags from ALU.

Otherwise, the value sent to the Register File would be result from ALU execution, and the new flags would be pulled from ALU.

If `UseImmediate` is 1, then MUX gate controlling the ALU `C` input would return the immediate value from signed ImmediateExtend. Otherwise the value of register `OutDataC` is used.

Now, the ALU execution result is returned back to the Register File, and if no `LoadUpperImmediate` signal was received, the result goes to the Register, whose address was specified in the Instruction.

Together with the result, New Flags are sent, and in case of an arithmetic operation.

Finally, the new clock cycle begins, the Program Counter is increased, and we start from the beginning.

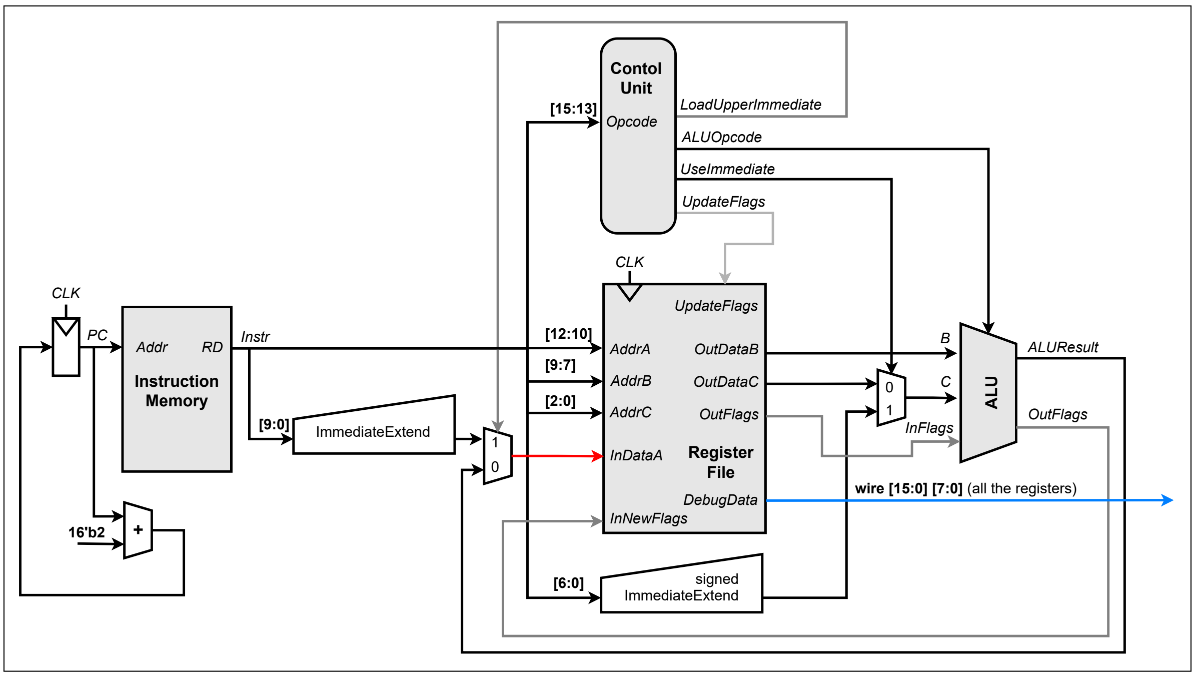

Figure 14: Diagram of the RISC-V module.

9. Conclusion

In conclusion, our project involved customizing the RISC-V architecture to achieve our specific goals, enabling the successful implementation of register loading, multiplication, and division using combinational logic. While we have made significant progress in understanding and implementing the hardware aspects of these operations, there is still potential for further optimization to improve their speed. It's important to note that optimizing the propagation time of multiplication and division may result in an increased number of gates. Finding a balance between faster operations and efficient use of hardware resources is crucial. Our project represents an initial exploration into hardware-level operations.

References

[1] [Online]. Available: https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html

[2] [Online]. Available: https://github.com/MKrekker/SINGLE-CYCLE-RISC-V

[3] [Online]. Available: https://coertvonk.com/hw/building-math-circuits/faster-parameterized-multiplier-in-verilog-30774

[4] [Online]. Available: https://coertvonk.com/hw/building-math-circuits/parameterized-divider-in-verilog-30776

[5] Github Copilot was used assist the code implementation. [Online]. Available: https://github.com/features/copilot